# Linux驱动\_按键与中断

## 1.ARM架构的Linux中断体系

### 1.1 一些常用的知识

#### 1.1.1 中断向量表位置

在ARM V4及V4T以后的大部分处理器中，中断向量表的基地址可以有两个位置：一个是0，另一个是0xffff0000。可以通过CP15协处理器c1寄存器中V 位(bit[13])控制。V和中断向量表的对应关系如下：

　　V**=**0 ～ 0x00000000**~**0x0000001C

　　V**=**1 ～ 0xffff0000**~**0xffff001C

#### 1.1.2 中断模式的stack准备

ARM处理器有多种processor mode，例如user mode（用户空间的AP所处于的模式）、supervisor mode（即SVC mode，大部分的内核态代码都处于这种mode）、IRQ mode（发生中断后，处理器会切入到该mode）等。对于linux kernel，其中断处理处理过程中，ARM 处理器大部分都是处于SVC mode。但是，实际上产生中断的时候，ARM处理器实际上是进入IRQ mode，因此在进入真正的IRQ异常处理之前会有一小段IRQ mode的操作，之后会进入SVC mode进行真正的IRQ异常处理。由于IRQ mode只是一个过度，因此IRQ mode的栈很小，只有12个字节，具体如下：

struct stack **{**

    u32 irq**[**3**];**

    u32 abt**[**3**];**

    u32 und**[**3**];**

**}** \_\_\_\_cacheline\_aligned**;**

static struct stack stacks**[**NR\_CPUS**];**

#### 1.1.3 SVC模式的stack准备

我们经常说进程的用户空间和内核空间，对于一个应用程序而言，可以运行在用户空间，也可以通过系统调用进入内核空间。在用户空间，使用的是用户栈，也就是我们软件工程师编写用户空间程序的时候，保存局部变量的stack。陷入内核后，当然不能用用户栈了，这时候就需要使用到内核栈。所谓内核栈其实就是处于SVC mode时候使用的栈。在linux最开始启动的时候，系统只有一个进程（更准确的说是kernel thread），就是PID等于0的那个进程，叫做swapper进程（或者叫做idle进程）。该进程的内核栈是静态定义的，如下：

union thread\_union init\_thread\_union \_\_init\_task\_data **=**

**{** INIT\_THREAD\_INFO**(**init\_task**)** **};**

union thread\_union **{**

    struct thread\_info thread\_info**;**

    unsigned long stack**[**THREAD\_SIZE**/sizeof(**long**)];**

对于ARM平台，THREAD\_SIZE是8192个byte，因此占据两个page frame。随着初始化的进行，Linux kernel会创建若干的内核线程，而在进入用户空间后，user space的进程也会创建进程或者线程。Linux kernel在创建进程（包括用户进程和内核线程）的时候都会分配一个（或者两个，和配置相关）page frame，具体代码如下：

static struct task\_struct **\***dup\_task\_struct**(**struct task\_struct **\***orig**)**

**{**

**......**

    ti **=** alloc\_thread\_info\_node**(**tsk**,** node**);**

**if** **(!**ti**)**

**goto** free\_tsk**;**

**......**

**}**

底部是struct thread\_info数据结构，顶部（高地址）就是该进程的内核栈。当进程切换的时候，整个硬件和软件的上下文都会进行切换，这里就包括了svc mode的sp寄存器的值被切换到调度算法选定的新的进程的内核栈上来。

### 1.2 中断的入口函数

#### 1.2.1 中断向量表

**.**equ stubs\_offset**,** \_\_vectors\_start **+** 0x200 **-** \_\_stubs\_start

**.**globl \_\_vectors\_start

\_\_vectors\_start**:**

swi SYS\_ERROR0 //系统复位中断

b vector\_und **+** stubs\_offset //未定义指令

ldr pc**,** **.**LCvswi **+** stubs\_offset //swi中断

b vector\_pabt **+** stubs\_offset //指令预取终止

b vector\_dabt **+** stubs\_offset //数据访问终止

b vector\_addrexcptn **+** stubs\_offset //保留

b vector\_irq **+** stubs\_offset //irq中断

b vector\_fiq **+** stubs\_offset //fiq中断

**.**globl \_\_vectors\_end

\_\_vectors\_end

这里以irq中断为例介绍，当发生irq中断时，硬件会帮忙自动执行b vector\_irq + stubs\_offset跳转到irq处理函数。  
在了解irq处理函数之前，先来分析一下中断向量表的搬移函数。

#define CONFIG\_VECTORS\_BASE 0xffff0000

void \_\_init early\_trap\_init**(**void**)**

**{**

unsigned long vectors **=** CONFIG\_VECTORS\_BASE**;**

extern char \_\_stubs\_start**[],** \_\_stubs\_end**[];**

extern char \_\_vectors\_start**[],** \_\_vectors\_end**[];**

extern char \_\_kuser\_helper\_start**[],** \_\_kuser\_helper\_end**[];**

int kuser\_sz **=** \_\_kuser\_helper\_end **-** \_\_kuser\_helper\_start**;**

memcpy**((**void **\*)**vectors**,** \_\_vectors\_start**,** \_\_vectors\_end **-** \_\_vectors\_start**);**

memcpy**((**void **\*)**vectors **+** 0x200**,** \_\_stubs\_start**,** \_\_stubs\_end **-** \_\_stubs\_start**);**

memcpy**((**void **\*)**vectors **+** 0x1000 **-** kuser\_sz**,** \_\_kuser\_helper\_start**,** kuser\_sz**);**

memcpy**((**void **\*)**KERN\_SIGRETURN\_CODE**,** sigreturn\_codes**,**

**sizeof(**sigreturn\_codes**));**

flush\_icache\_range**(**vectors**,** vectors **+** PAGE\_SIZE**);**

modify\_domain**(**DOMAIN\_USER**,** DOMAIN\_CLIENT**);**

**}**

该函数将\_\_vectors\_start到\_\_vectors\_end之间的代码拷贝到0xffff0000，  
 \_\_stubs\_start到\_\_stubs\_end之间的代码拷贝到0xffff0000+512。

#### 1.2.2 中断处理函数

##### 1.2.2.1 中断处理函数代码

还是以irq中断为例来分析，上面的b vector\_irq + stubs\_offset会跳转到vector\_stub irq, IRQ\_MODE, 4处。\_\_stubs\_end 至 \_\_stubs\_start之间是异常处理的位置，vector\_und、vector\_pabt、vector\_irq、vector\_fiq都在它们中间。

**.**globl \_\_stubs\_start

\_\_stubs\_start**:**

vector\_stub irq**,** IRQ\_MODE**,** 4

**.**long \_\_irq\_usr @ 0 **(**USR\_26 **/** USR\_32**)**

**.**long \_\_irq\_invalid @ 1 **(**FIQ\_26 **/** FIQ\_32**)**

**.**long \_\_irq\_invalid @ 2 **(**IRQ\_26 **/** IRQ\_32**)**

**.**long \_\_irq\_svc @ 3 **(**SVC\_26 **/** SVC\_32**)**

**.**long \_\_irq\_invalid @ 4

**.**long \_\_irq\_invalid @ 5

**.**long \_\_irq\_invalid @ 6

**.**long \_\_irq\_invalid @ 7

**.**long \_\_irq\_invalid @ 8

**.**long \_\_irq\_invalid @ 9

**.**long \_\_irq\_invalid @ a

**.**long \_\_irq\_invalid @ b

**.**long \_\_irq\_invalid @ c

**.**long \_\_irq\_invalid @ d

**.**long \_\_irq\_invalid @ e

**.**long \_\_irq\_invalid @ f

**...**

**.**globl \_\_stubs\_end

\_\_stubs\_end**:**

##### 1.2.2.2 stubs\_offset的作用

由于中断向量表和中断处理函数进行了移动，所以不能直接用b vector\_irq跳转到中断处理函数，需要计算出搬移后两个命令之间的偏移量，才能正确的进行跳转。搬移后vector\_irq符号的地址：

CONFIG\_VECTORS\_BASE **+** 0x200 **+** vector\_irq**(**编译地址**)** **-** \_\_stubs\_start**(**编译地址**)**

执行中断异常时，指令地址应该为：

CONFIG\_VECTORS\_BASE **+** （irq\_PC **-** \_\_vectors\_start）

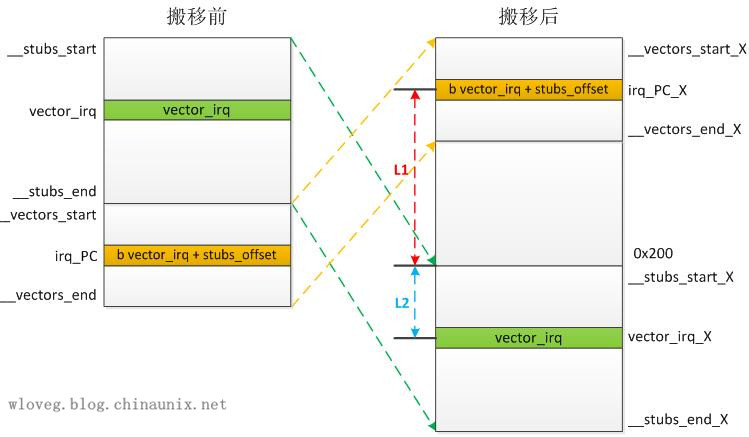
所以两条指令之间搬移后的偏移地址为：

（CONFIG\_VECTORS\_BASE **+** 0x200 **+** （vector\_irq**(**编译地址**)** **-** \_\_stubs\_start**(**编译地址**)**）） **-** （CONFIG\_VECTORS\_BASE **+** （irq\_PC**)** **–**

\_\_vectors\_start（编译地址）））

**=** （0x200 **+** （vector\_irq**(**编译地址**)** **-** \_\_stubs\_start**(**编译地址**)**）） **-** （irq\_PC **-** \_\_vectors\_start（编译地址））

**=** \_\_vectors\_start **+** 0x200 **-** \_\_stubs\_start **+** vector\_irq **-** irq\_PC



令stubs\_offset = \_\_vectors\_start + 0x200 - \_\_stubs\_start,则offset = vector\_irq + stubs\_offset -irq\_PC，所以中断入口点为  
b vector\_irq + stubs\_offset，其中减去irq\_PC是由汇编器在编译时完成的。

##### 1.2.2.3 vector\_irq

切入正题，我们目前并未发现关于vector\_irq的定义，vector\_stub irq, IRQ\_MODE, 4语句展开后就是vector\_irq，相关宏定义如下：

**.**macro vector\_stub**,** name**,** mode**,** correction**=**0

**.**align 5

vector\_\name**:**

//当发生IRQ中断的时候，lr中保存了发生中断的PC＋4，

//如果减去4的话，得到的就是发生中断那一点的PC值。

**.if** \correction

sub lr**,** lr**,** #\correction

**.**endif

//当前是IRQ mode,SP\_irq在初始化的时候已经设定(12个字节)

//在irq mode的stack上，依次保存了发生中断那一点的r0值、PC值以及CPSR值

//(具体操作是通过spsr进行的，其实硬件已经帮我们保存了CPSR到SPSR中了)

//为何要保存r0值？因为随后的代码要使用r0寄存器，

//因此我们要把r0放到栈上，只有这样才能完完全全恢复硬件现场。

stmia sp**,** **{**r0**,** lr**}** @ save r0**,** lr

mrs lr**,** spsr

str lr**,** **[**sp**,** #8**]** @ save spsr

//可怜的IRQ mode稍纵即逝，这段代码就是准备将ARM推送到SVC mode

//如何准备？其实就是修改SPSR的值，SPSR不是CPSR，

//不会引起processor mode的切换（毕竟这一步只是准备而已）。

mrs r0**,** cpsr

eor r0**,** r0**,** #**(**\mode **^** SVC\_MODE**)**

msr spsr\_cxsf**,** r0

//很多异常处理的代码返回的时候都是使用了stack相关的操作，这里没有

//“movs    pc, lr ”指令除了字面上意思（把lr的值付给pc），

//还有一个隐含的操作（movs中‘s’的含义）：把SPSR copy到CPSR，从而实现了模式的切换。

**and** lr**,** lr**,** #0x0f  //获取被中断前，处理器所处的模式

mov r0**,** sp //让r0寄存器指向中断模式下堆栈的基地址

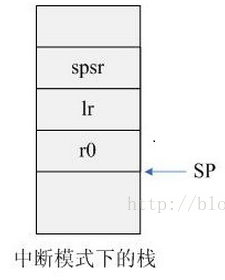
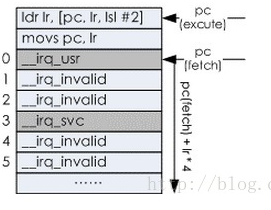
ldr lr**,** **[**pc**,** lr**,** lsl #2**]** //获取跳转的位置

movs pc**,** lr @ branch to handler in SVC mode

**.**endm

当发生IRQ中断时，CPU自动完成的工作有：  
(1)IRQ模式下的R14(LR)寄存器保存当前PC值+4或者+8。  
(2)将CPSR的值复制到IRQ模式的SPSR寄存器。  
(3)将CPSR的工作模式位设置为IRQ模式  
(4)PC值跳到中断向量表IRQ的位置。

在vector\_irq中，主要完成如下工作(切换到SVC模式)：  
(1)IRQ的栈保存了spsr,lr,r0寄存器值，也就是保存了发生中断时的CPSR,PC,r0寄存器的值。  
(2)修改SPSR，准备从CPSR切换到SVC模式。  
(3)获取发生中断前，处理器是处于USER还是SVC模式，根据处理器模式的不同，分别跳转到\_\_irq\_usr和\_\_irq\_svc两个分支。并将r0寄存器指向IRQ模式下的SP寄存器。

寄存器结构体定义如下：

struct pt\_regs **{**

long uregs**[**18**];**

**};**

#define ARM\_cpsr uregs[16]

#define ARM\_pc uregs[15]

#define ARM\_lr uregs[14]

#define ARM\_sp uregs[13]

#define ARM\_ip uregs[12]

#define ARM\_fp uregs[11]

#define ARM\_r10 uregs[10]

#define ARM\_r9 uregs[9]

#define ARM\_r8 uregs[8]

#define ARM\_r7 uregs[7]

#define ARM\_r6 uregs[6]

#define ARM\_r5 uregs[5]

#define ARM\_r4 uregs[4]

#define ARM\_r3 uregs[3]

#define ARM\_r2 uregs[2]

#define ARM\_r1 uregs[1]

#define ARM\_r0 uregs[0]

#define ARM\_ORIG\_r0 uregs[17]

##### 1.2.2.4 \_\_irq\_usr

当发生中断的时候，代码运行在用户空间,则会运行\_\_irq\_usr。

**.**align 5

\_\_irq\_usr**:**

usr\_entry

get\_thread\_info tsk

irq\_handler

mov why**,** #0

b ret\_to\_user

(1)usr\_entry 保存发生中断时候的现场。所谓保存现场其实就是把发生中断那一刻的硬件上下文（各个寄存器）保存在了SVC mode的stack上。

include**/**asm**-**arm**/**asm**-**offsets**.**h**:**40**:**#define S\_PC 60 /\* offsetof(struct pt\_regs, ARM\_pc) @ \*/

**.**macro usr\_entry

//进入svc模式，sp切换到svc模式的sp

//S\_FRAME\_SIZE为18\*4=72，也就是需要保存18个寄存器

sub sp**,** sp**,** #S\_FRAME\_SIZE

//svc栈压入r1～r12

//stmib中的ib表示increment before，因此，在压入R1的时候，stack pointer会先增加4，重要是预留r0的位置

//stmib没有！的修饰符，表示压栈完成后并不会真正更新stack pointer，因此sp保持原来的值

stmib sp**,** **{**r1 **-** r12**}**

//这里r0指向了irq stack

//因此，r1是中断时候的r0值，r2是中断现场的PC值，r3是中断现场的CPSR值

ldmia r0**,** **{**r1 **-** r3**}**

//r0 = sp+60,将r0指向ARM\_pc

add r0**,** sp**,** #S\_PC

//r4保存-1

mov r4**,** #**-**1

//保存中断那一刻的r0 ,保存在ARM\_r0

str r1**,** **[**sp**]**

//将r2 r2 r3依次保存到ARM\_pc ARM\_cpsr ARM\_ORIG\_r0

stmia r0**,** **{**r2 **-** r4**}**

//我们保存的是发生中断那一刻（对于本节，这是当时user mode的sp和lr），指令中的^符号表示访问user mode的寄存器。

stmdb r0**,** **{**sp**,** lr**}^**

//alignment\_trap在配置CONFIG\_ALIGNMENT\_TRAP时有效，如果开启了该选项，中断处理中将支持对齐跟踪

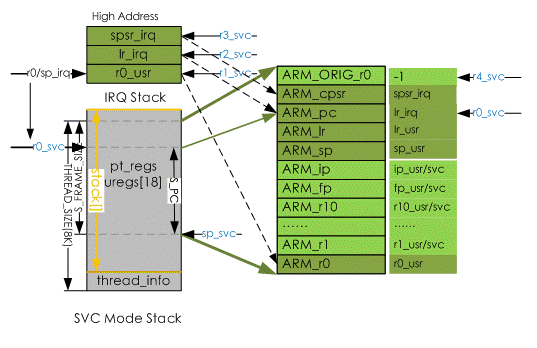
alignment\_trap r0

//zero\_fp用来设置fp栈帧寄存器为0

zero\_fp

**.**endm

usr\_entry函数操作过程示意如下图所示：



(2)get\_thread\_info

根据当前sp指针，将该指针最右边13位清0，获得当前任务的thread\_info。

**.**macro get\_thread\_info**,** rd

mov \rd**,** sp**,** lsr #13

mov \rd**,** \rd**,** lsl #13

**.**endm

其中，进程栈的栈底，存储着该进程的基本信息如id 调度优先级等信息。  
栈顶指针低13位清0，显然高位是base地址，低13位看来是整个进程栈的偏移地址，由此可以得到栈的base地址。

tsk .req r9 @ current thread\_info

thread\_info和栈的关系。

union thread\_union **{**

    struct thread\_info thread\_info**;**  /\*线程属性\*/

    unsigned long stack**[**THREAD\_SIZE**/sizeof(**long**)];**  /\*栈\*/

**};**

(3)irq\_handler

中断服务例程，核心处理内容，后面再进行分析。

(4) mov why**,** #0

r8寄存器赋值为0。

why .req r8 @ Linux syscall **(!=** 0**)**

(5) b ret\_to\_user

返回到用户模式，中断之前的状态。

ENTRY**(**ret\_to\_user**)**

ret\_slow\_syscall**:**

disable\_irq @ disable interrupts

//从任务的TI\_FLAGS标志判断是否需要处理抢占或者信号。

ldr r1**,** **[**tsk**,** #TI\_FLAGS**]**

tst r1**,** #\_TIF\_WORK\_MASK

//处理抢占或者信号

bne work\_pending

//没有抢占或者信号需要处理，或者已经处理完毕，开始退回用户态

no\_work\_pending**:**

//在返回用户态前，处理各个体系结构的钩子

arch\_ret\_to\_user r1**,** lr

//寄存器恢复,回到中断之前的状态

//获取中断时的cpsr寄存器值

ldr r1**,** **[**sp**,** #S\_PSR**]** @ get calling cpsr

//获取中断时的pc值,这里会更新sp的值，sp = sp+60

ldr lr**,** **[**sp**,** #S\_PC**]!** @ get pc

//将发生中断时的cpsr的值赋给spsr\_svc

msr spsr\_cxsf**,** r1 @ save in spsr\_svc

//恢复r0-lr寄存器值，并将spsr的值赋cpsr

ldmdb sp**,** **{**r0 **-** lr**}^** @ get calling r1 **-** lr

//nop 不做任何事情

mov r0**,** r0

//恢复内核栈到中断产生之前的位置。

//sp = sp+72-60 = sp+12

//目前sp的位置为sp+60

add sp**,** sp**,** #S\_FRAME\_SIZE **-** S\_PC

//返回中断前的代码处

movs pc**,** lr @ **return** **&** move spsr\_svc into cpsr

##### 1.2.2.5 \_\_irq\_svc

当发生中断的时候，代码运行在内核空间,则会运行\_\_irq\_svc。

**.**align 5

\_\_irq\_svc**:**

svc\_entry

irq\_handler

ldr r0**,** **[**sp**,** #S\_PSR**]** @ irqs are already disabled

msr spsr\_cxsf**,** r0

ldmia sp**,** **{**r0 **-** pc**}^** @ load r0 **-** pc**,** cpsr

(1)svc\_entry

保存发生中断那一刻的现场保存在内核栈上，和usr模式处理中断不太一样的是，usr模式发生中断时，cpu模式变化为usr --> irq --> svc，而svc模式发生中断后，cpu模式变化为svc --> irq --> svc。

include**/**asm**-**arm**/**asm**-**offsets**.**h**:**38**:**#define S\_SP 52 /\* offsetof(struct pt\_regs, ARM\_sp) @ \*/

**.**macro svc\_entry

//分配栈空间保存18个寄存器

sub sp**,** sp**,** #S\_FRAME\_SIZE

//sp先增加4，保存r1-r12到对应的栈中

stmib sp**,** **{**r1 **-** r12**}**

//恢复r0寄存器的内容到r1-r3

//r1是中断时候的r0值，r2是中断现场的PC值，r3是中断现场的CPSR值

ldmia r0**,** **{**r1 **-** r3**}**

//r5 = sp+52 ARM\_sp寄存器位置

add r5**,** sp**,** #S\_SP

//r4 = -1

mov r4**,** #**-**1

//r0 = sp+72

//发生中断时sp指针的位置

//因为是从svc模式进入IRQ，再回到svc模式，所以发生中断时的sp指针就为sp+72

add r0**,** sp**,** #S\_FRAME\_SIZE

//保存真正的r0到sp

str r1**,** **[**sp**]**

//将lr寄存器的值暂存在r1

mov r1**,** lr

//保存中断时的 SP PC CPSR 值到栈中

stmia r5**,** **{**r0 **-** r4**}**

**.**endm

(2) irq\_handler

中断处理函数，在后面进行分析。

(3) 剩余部分

将spsr\_cxsf的内容读到栈的ARM\_CPSR位置处，最后恢复所有的寄存器。

ldr r0**,** **[**sp**,** #S\_PSR**]** @ irqs are already disabled

msr spsr\_cxsf**,** r0

ldmia sp**,** **{**r0 **-** pc**}^** @ load r0 **-** pc**,** cpsr

include**/**asm**-**arm**/**asm**-**offsets**.**h**:**41**:**#define S\_PSR 64 /\* offsetof(struct pt\_regs, ARM\_cpsr) @ \*/

##### 1.2.2.6 irq\_handler

获取中断号，并跳转到C语言写的中断处理函数中。

**.**macro irq\_handler

//这里为空函数

get\_irqnr\_preamble r5**,** lr

//获取本次中断的硬件中断号,保存在r0中

1**:** get\_irqnr\_and\_base r0**,** r6**,** r5**,** lr

//如果上面函数设置了Z flag，则r1=sp，

//如果上面找到了中断号，Z flag = 1

movne r1**,** sp

//如果Z flag被设置了，则设置lr为标号1

//也就是asm\_do\_IRQ的返回地址

adrne lr**,** 1b

//如果Z flag被设置了，则跳到C语言的中断处理函数

//r0=中断号 r1=struct pt\_regs

bne asm\_do\_IRQ

**.**endm

获取中断号的代码如下：

**.**macro get\_irqnr\_and\_base**,** irqnr**,** irqstat**,** base**,** tmp

//对应 r0, r6, r5, lr

//mov r5, #S3C24XX\_VA\_IRQ

mov \base**,** #S3C24XX\_VA\_IRQ

//获取INTPND寄存器的数据 存入 r6

ldr \irqstat**,** **[** \base**,** #INTPND **]**

//测试r6的值，看是否为0

teq \irqstat**,** #0

//如果为0 就说明没有中断发生,INTPND寄存器中没有置1的位跳转到1002标号处

beq 1002f

//如果不为0，说明INTPND有位置1，这里读取INTOFFSET寄存器的内容到r0中

ldr \irqnr**,** **[** \base**,** #INTOFFSET **]**

//给lr赋值1

mov \tmp**,** #1

//测试r6和1<<r0 进行逻辑按位与操作

tst \irqstat**,** \tmp**,** lsl \irqnr

//如果上面为相等，那么就说明找到了中断号，就直接调到1001标号

bne 1001f

//如果不等，则继续执行,要自己计算中断号

//r0 = 0

mov \irqnr**,** #0

movs \tmp**,** \irqstat**,** lsl#16

addeq \irqnr**,** \irqnr**,** #16

moveq \irqstat**,** \irqstat**,** lsr#16

tst \irqstat**,** #0xff

addeq \irqnr**,** \irqnr**,** #8

moveq \irqstat**,** \irqstat**,** lsr#8

tst \irqstat**,** #0xf

addeq \irqnr**,** \irqnr**,** #4

moveq \irqstat**,** \irqstat**,** lsr#4

tst \irqstat**,** #0x3

addeq \irqnr**,** \irqnr**,** #2

moveq \irqstat**,** \irqstat**,** lsr#2

tst \irqstat**,** #0x1

addeq \irqnr**,** \irqnr**,** #1

1001**:**

//这里r0 ＝ r0 + IRQ\_EINT0，这里便得到了中断号

adds \irqnr**,** \irqnr**,** #IRQ\_EINT0

1002**:**

@@ exit here**,** Z flag unset **if** IRQ

**.**endm

##### 1.2.2.7 asm\_do\_IRQ

在irq\_desc结构体中找到对应的处理函数，并执行。后面则是需要分析irq\_desc结构体是如何被初始化的。

asmlinkage void \_\_exception asm\_do\_IRQ**(**unsigned int irq**,** struct pt\_regs **\***regs**)**

**{**

//记录中断现场

struct pt\_regs **\***old\_regs **=** set\_irq\_regs**(**regs**);**

struct irq\_desc **\***desc **=** irq\_desc **+** irq**;**

**if** **(**irq **>=** NR\_IRQS**)**

desc **=** **&**bad\_irq\_desc**;**

//irq\_enter就是告诉系统,现在正在处理中断的上半部分工作,不可以进行调度

irq\_enter**();**

//调用相应的处理函数

desc\_handle\_irq**(**irq**,** desc**);**

//AT91使用

irq\_finish**(**irq**);**

irq\_exit**();**

//恢复中断现场

set\_irq\_regs**(**old\_regs**);**

**}**

### 1.3 中断处理函数的初始化

start\_kernel会调用init\_IRQ()开始开始中断处理函数的初始化。

void \_\_init init\_IRQ**(**void**)**

**{**

int irq**;**

**for** **(**irq **=** 0**;** irq **<** NR\_IRQS**;** irq**++)**

irq\_desc**[**irq**].**status **|=** IRQ\_NOREQUEST **|** IRQ\_NOPROBE**;**

init\_arch\_irq**();**

**}**

其中init\_arch\_irq为s3c24xx\_init\_irq。  
s3c24xx\_init\_irq主要用于清空相关的中断寄存器，并设置中断的操作函数和处理函数。具体是如何调用这些初始化的函数进行中断响应，后面会解释。

void \_\_init s3c24xx\_init\_irq**(**void**)**

**{**

unsigned long pend**;**

unsigned long last**;**

int irqno**;**

int i**;**

//1.清空相应的寄存器

//清空EINTPEND寄存器

last **=** 0**;**

**for** **(**i **=** 0**;** i **<** 4**;** i**++)** **{**

pend **=** \_\_raw\_readl**(**S3C24XX\_EINTPEND**);**

**if** **(**pend **==** 0 **||** pend **==** last**)**

**break;**

\_\_raw\_writel**(**pend**,** S3C24XX\_EINTPEND**);**

printk**(**"irq: clearing pending ext status %08x\n"**,** **(**int**)**pend**);**

last **=** pend**;**

**}**

//清空INTPND寄存器

last **=** 0**;**

**for** **(**i **=** 0**;** i **<** 4**;** i**++)** **{**

pend **=** \_\_raw\_readl**(**S3C2410\_INTPND**);**

**if** **(**pend **==** 0 **||** pend **==** last**)**

**break;**

\_\_raw\_writel**(**pend**,** S3C2410\_SRCPND**);**

\_\_raw\_writel**(**pend**,** S3C2410\_INTPND**);**

printk**(**"irq: clearing pending status %08x\n"**,** **(**int**)**pend**);**

last **=** pend**;**

**}**

//清空SUBSRCPND寄存器

last **=** 0**;**

**for** **(**i **=** 0**;** i **<** 4**;** i**++)** **{**

pend **=** \_\_raw\_readl**(**S3C2410\_SUBSRCPND**);**

**if** **(**pend **==** 0 **||** pend **==** last**)**

**break;**

printk**(**"irq: clearing subpending status %08x\n"**,** **(**int**)**pend**);**

\_\_raw\_writel**(**pend**,** S3C2410\_SUBSRCPND**);**

last **=** pend**;**

**}**

//2.设置中断处理函数

//handle\_level\_irq 用于电平触发中断的流控处理；

//handle\_edge\_irq 用于边沿触发中断的流控处理；

//set\_irq\_chip 用于设置这些外部中断的触发方式（电平触发，边沿触发），使能中断，禁止中断

//set\_irq\_handler 设置中断处理函数

//set\_irq\_chained\_handler 设置串行中断处理函数，也就是一个中断号中还有其他子中断号，需要进一步处理。设置串行时，会开启该中断源，这样下面的子中断才会得到处理

**for** **(**irqno **=** IRQ\_EINT4t7**;** irqno **<=** IRQ\_ADCPARENT**;** irqno**++)** **{**

**switch** **(**irqno**)** **{**

**case** IRQ\_EINT4t7**:**

**case** IRQ\_EINT8t23**:**

**case** IRQ\_UART0**:**

**case** IRQ\_UART1**:**

**case** IRQ\_UART2**:**

**case** IRQ\_ADCPARENT**:**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_level\_chip**);**

set\_irq\_handler**(**irqno**,** handle\_level\_irq**);**

**break;**

**case** IRQ\_RESERVED6**:**

**case** IRQ\_RESERVED24**:**

/\* no IRQ here \*/

**break;**

**default:**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_chip**);**

set\_irq\_handler**(**irqno**,** handle\_edge\_irq**);**

//该函数主要是为特定的中断设置相应的状态标记，

//而这里我们调用它的目的就是清掉IRQ\_NOREQUEST标记，

//告诉系统该中断已经可以被申请使用了，中断在申请的时候会查看是否有IRQ\_NOREQUEST标记，如有则表明该中断还不能使用。而初始化的时候所有的中断都有这个标记。

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**}**

/\* setup the cascade irq handlers \*/

set\_irq\_chained\_handler**(**IRQ\_EINT4t7**,** s3c\_irq\_demux\_extint4t7**);**

set\_irq\_chained\_handler**(**IRQ\_EINT8t23**,** s3c\_irq\_demux\_extint8**);**

set\_irq\_chained\_handler**(**IRQ\_UART0**,** s3c\_irq\_demux\_uart0**);**

set\_irq\_chained\_handler**(**IRQ\_UART1**,** s3c\_irq\_demux\_uart1**);**

set\_irq\_chained\_handler**(**IRQ\_UART2**,** s3c\_irq\_demux\_uart2**);**

set\_irq\_chained\_handler**(**IRQ\_ADCPARENT**,** s3c\_irq\_demux\_adc**);**

/\* external interrupts \*/

**for** **(**irqno **=** IRQ\_EINT0**;** irqno **<=** IRQ\_EINT3**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (ext int)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_eint0t4**);**

set\_irq\_handler**(**irqno**,** handle\_edge\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**for** **(**irqno **=** IRQ\_EINT4**;** irqno **<=** IRQ\_EINT23**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (extended s3c irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irqext\_chip**);**

set\_irq\_handler**(**irqno**,** handle\_edge\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

/\* register the uart interrupts \*/

irqdbf**(**"s3c2410: registering external interrupts\n"**);**

**for** **(**irqno **=** IRQ\_S3CUART\_RX0**;** irqno **<=** IRQ\_S3CUART\_ERR0**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (s3c uart0 irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_uart0**);**

set\_irq\_handler**(**irqno**,** handle\_level\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**for** **(**irqno **=** IRQ\_S3CUART\_RX1**;** irqno **<=** IRQ\_S3CUART\_ERR1**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (s3c uart1 irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_uart1**);**

set\_irq\_handler**(**irqno**,** handle\_level\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**for** **(**irqno **=** IRQ\_S3CUART\_RX2**;** irqno **<=** IRQ\_S3CUART\_ERR2**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (s3c uart2 irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_uart2**);**

set\_irq\_handler**(**irqno**,** handle\_level\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

**for** **(**irqno **=** IRQ\_TC**;** irqno **<=** IRQ\_ADC**;** irqno**++)** **{**

irqdbf**(**"registering irq %d (s3c adc irq)\n"**,** irqno**);**

set\_irq\_chip**(**irqno**,** **&**s3c\_irq\_adc**);**

set\_irq\_handler**(**irqno**,** handle\_edge\_irq**);**

set\_irq\_flags**(**irqno**,** IRQF\_VALID**);**

**}**

irqdbf**(**"s3c2410: registered interrupt handlers\n"**);**

**}**

### 1.4 中断注册

在驱动程序需要调用request\_irq为irq注册实际的中断处理函数。例如：

request\_irq**(**IRQ\_EINT0**,** buttons\_irq**,** IRQT\_BOTHEDGE**,** "S2"**,** **&**pins\_desc**[**0**]);**

函数原型：

int request\_irq**(**unsigned int irq**,** irq\_handler\_t handler**,**unsigned long irqflags**,** const char **\***devname**,** void **\***dev\_id**)**

**irq**是要申请的硬件中断号。  
**handler**是向系统注册的中断处理函数，是一个回调函数，中断发生时，系统调用这个函数，dev\_id参数将被传递给它。  
**irqflags**是中断处理的属性，若设置了IRQF\_DISABLED，则表示中断处理程序是快速处理程序，快速处理程序被调用时屏蔽所有中断，慢速处理程序不屏蔽；若设置了IRQF\_SHARED，则表示多个设备共享中断，若设置了IRQF\_SAMPLE\_RANDOM，表示对系统熵有贡献，对系统获取随机数有好处。(这几个flag是可以通过或的方式同时使用的）。  
**devname**设置中断名称，通常是设备驱动程序的名称  在cat /proc/interrupts中可以看到此名称。  
**dev\_id**在中断共享时会用到，一般设置为这个设备的设备结构体或者NULL。  
request\_irq()返回0表示成功，返回-INVAL表示中断号无效或处理函数指针为NULL，返回-EBUSY表示中断已经被占用且不能共享。

int request\_irq**(**unsigned int irq**,** irq\_handler\_t handler**,**

unsigned long irqflags**,** const char **\***devname**,** void **\***dev\_id**)**

**{**

struct irqaction **\***action**;**

int retval**;**

//shared interrupts must pass in a real dev-ID,

**if** **((**irqflags **&** IRQF\_SHARED**)** **&&** **!**dev\_id**)**

**return** **-**EINVAL**;**

**if** **(**irq **>=** NR\_IRQS**)**

**return** **-**EINVAL**;**

//该中断号已经被使用，并且未共享

**if** **(**irq\_desc**[**irq**].**status **&** IRQ\_NOREQUEST**)**

**return** **-**EINVAL**;**

**if** **(!**handler**)**

**return** **-**EINVAL**;**

//分配了一个结构，结构中的成员指向传递进来的参数

action **=** kmalloc**(sizeof(**struct irqaction**),** GFP\_ATOMIC**);**

**if** **(!**action**)**

**return** **-**ENOMEM**;**

action**->**handler **=** handler**;**

action**->**flags **=** irqflags**;**

cpus\_clear**(**action**->**mask**);**

action**->**name **=** devname**;**

action**->**next **=** **NULL;**

action**->**dev\_id **=** dev\_id**;**

select\_smp\_affinity**(**irq**);**

//设置irq

//1)irq\_des[irq] 已irq为下标找到数组项

//2)在irq\_des[irq]链表里面加入传递进来的参数action

//3)desc->chip->settype()设置为中断引脚

//4)desc->chip->startup / desc->chip->enable 使能中断

retval **=** setup\_irq**(**irq**,** action**);**

**if** **(**retval**)**

kfree**(**action**);**

**return** retval**;**

**}**

从源码看request\_irq会使能该中断。

### 1.5 外部中断处理流程分析

#### 1.5.1 外部中断0

1.5.1.1 外部中断0的初始化

1.5.1.2 外部中断0的注册

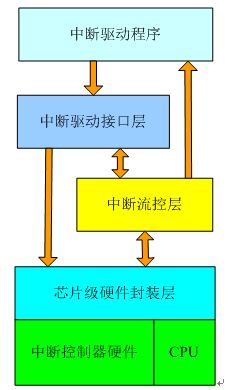
1.5.1.3 外部中断的响应

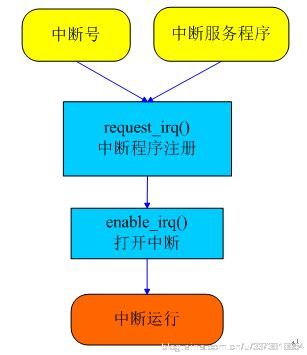
对于外部中断0，在其struct irq\_desc的handle域为handle\_edge\_irq函数。

中断处理流程

顶半部和底半部

一开始内核的开发者们把3种中断类型区分出来（电平中断、边缘中断、简易中断），后来又针对某些需要回应eoi（end of interrupt）的中断控制器，加入了fast eoi type，针对smp加入了per cpu type。把这些不同的中断类型抽象出来后，成为了中断子系统的流控层。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=c91863e60102w48u&url=http://album.sina.com.cn/pic/003Gkcp8gy6Yr0p7UTP7d)

[](http://photo.blog.sina.com.cn/showpic.html#blogid=c91863e60102w48u&url=http://album.sina.com.cn/pic/003Gkcp8gy6Yr0MBuUIcc)